Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Andrew J. Hutton, *Steamballoon, Inc.*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

cpuidle—Do nothing, efficiently...

Venkatesh Pallipadi
Shaohua Li
Intel Open Source Technology Center
{venkatesh.pallipadi|shaohua.li}@intel.com

Adam Belay
Novell, Inc.
abelay@novell.com

Abstract

Most of the focus in Linux processor power management today has been on power managing the processor while it is active: `cpufreq`, which changes the processor frequency and/or voltage and manages the processor performance levels and power consumption based on processor load. Another dimension of processor power management is processor 'idling' power.

Almost all mobile processors in the marketplace today support the concept of multiple processor idle states with varying amounts of power consumed in those idle states. Each such state will have an entry-exit latency associated with it. In general, there is a lot of attention shifting towards idle platform power and new platforms/processors are supporting multiple idle states with different power and wakeup latency characteristics. This emphasis on idle power and different processors supporting different number of idle states and different ways of entering these states, necessitates the need for a generic Linux kernel framework to manage idle processors.

This paper covers `cpuidle`, an effort towards a generic processor idle management framework in Linux kernel. The goal is to have a clean interface for any processor hardware to make use of different processor idle levels and also provide abstraction between idle-drivers and idle-governors allowing independent development of drivers and governors. The target audiences are the developers who are keen to experiment with new idle governors on top of `cpuidle`, and developers who wants to use the `cpuidle` driver infrastructure in various architectures, and any one else who is keen to know about `cpuidle`.

1 Introduction

Almost all the mobile processors today support multiple idle states and the trend is spreading as processor power management and system power management gain importance for a variety of reasons.

In typical system usage models, processor(s) spend a lot of their time idling (like while you are reading this paper on your laptop, with your favorite pdf-reader). Thus any power saved when system is idle will have big returns in terms of battery life, heat generated in the system, need for cooling, etc.

But there is a trade-off between idling power and amount of state a processor saves and the amount of time it takes to enter and exit from this idle state. The idle enter-exit latency, if it is too high, may be visible with media applications like a DVD player. Such usage models will limit the usage of a particular idle state on the processor running this application, even though the idle state is power efficient. Similarly, if a processor idle state does not preserve the the contents of the processor's cache, some particular application which has some idle time may notice a performance degradation when this particular idle state is used.

In order to manage this trade-off effectively, the kernel needs to know the characteristics of all idle states and also should understand the currently running applications, and should take a well-informed decision about what idle state it wants to enter when processor goes to idle.

To do this effectively and cleanly, there is a preliminary requirement of having clean and simple interfaces. Such an interface can provide consistent information to the user and ease the innovation and development in the area of processor idle management.

`cpuidle` is a an effort in this direction and this paper provides insight into `cpuidle`. We start section 2 with a background on processor power management and idle states. Section 3 provides the design description of `cpuidle`. Section 4 talks about all the develop-

ments and advancements happening in `cpuidle` and some conclusions in section 5.

2 Background

2.1 Processor Power management

Processor power management can be broadly classified into two classes.

Processor active – various states a processor can be in while actively executing and retiring instructions. Processor frequency scaling, in which a processor can run at different frequencies and or voltages falls under this class. So does processor thermal throttling, where processor runs slower due to duty cycle throttling.

Linux `cpufreq`, extensively discussed in [4], [6], and [5], is a generic infrastructure that handles CPU frequency scaling.

Processor idle – various states a processor can be in while it is idle and not retiring any instructions. The states here differ in amount of power the processor consumes while being in that state and also the latency to enter-exit this low-power idle state. There may also be other differences like preserving the processor state across these idle states, etc. based on a specific processor. For example, a processor may only flush L1 cache in one idle state, but may flush L1 and L2 caches in another idle state. There can also be differences around when an idle state can be entered and what its impact will be on other logical or physical processors in the system.

2.2 Processor idle states

Currently, most of the processors in mobile and handheld segments support multiple idle states. The prime objective here is to provide a more power-efficient system with longer battery life or fewer cooling requirements. This feature is slowly moving up the chain into desktops and servers. This is much like processor frequency scaling which was mostly present in mobile processors a few years back, to most of the servers supporting that feature today. Recent EnergyStar idle power regulations [2] are tending to make this faster, making this feature more common across a range of systems.

2.3 Current Processor idle state support

Below is a short summary of current processor idle state management in Linux 2.6.21 [3].

ACPI based idle states For the remainder of this section we restrict our attention to idle state support as in i386 (and x86-64) architectures.

In i386 (and x86-64) architectures, there is support for ACPI-based [1] processor idle states. These states are referred to as C-states in ACPI terminology. Each of the ACPI C-states is characterised by its power consumption and wakeup latency, and also based on preservation of the processor state, while in this C-state. ACPI-based platforms will report processor idle capability to Linux using ACPI interfaces. A platform can dynamically change the number of C-states supported, based on different platform parameters such as whether it is running on battery or AC power.

The current Linux support for such idle states is fully embedded in the `drivers/acpi` directory along with all ACPI support code. Code here detects the C-states available at boot time, handles any changes to the number of C-states during run time, and has simplistic policy to choose a particular C-state to enter into whenever a CPU goes idle. This code includes various platform-specific bits, specific workarounds for platform ACPI bugs, and also a `/proc`-based interface exporting the C-state-related information to userspace.

Arch specific idle—i386 and x86-64 i386 and x86-64 (and also ia64) have some architecture-specific processor idle management that does not depend on ACPI. On i386 and x86-64, it includes support for `poll_idle`, `halt_idle`, and `mwait_idle`. `poll_idle` is a polling-based idle loop, which is not really power efficient, but will have very little wakeup overhead. `halt_idle` is based on the x86 `hlt` instruction, and `mwait_idle` is based on the `monitor mwait` pair of instructions. There are specific static rules regarding which of these idle routines will be used on any system, based on boot options and hardware capabilities. Further, boot options across x86 and x86-64 are not the same for these three idle routines.

Arch-specific idle—other architectures There are various other architectures that have their own

code for processor idle state management. This includes ia64 with PAL halt and PAL light halt, Power with nap and doze modes, and idle support for different platforms in the ARM architecture. Each of these types of support for idle states also comes with its own set of boot parameters and/or `/proc` or `/sys` interfaces to user-space.

Bottom line There is very little sharing of code and sharing of idle management policies across architectures. Processor idle state management and various boot options, etc., are duplicated; this results in code duplication and maintenance overhead. This, as well as the increasing focus on processor idle power in platforms, highlights the need for a generic processor idle framework in Linux kernel.

3 Basic `cpuidle` infrastructure

Figure 1 gives a high-level overview of the `cpuidle` architecture. The basic idea behind `cpuidle` is to separate the idle state management policies from hardware-specific idle state drivers. At this level, the `cpuidle` model has similarities with `cpufreq` [6].

3.1 `cpuidle` core

The `cpuidle` core provides a set of generic interfaces to manage processor idle features.

3.1.1 `cpuidle` data structures

A per-cpu `cpuidle_device` structure holds information about the number of idle states supported by each processor, information about each of those idle state (in an array of `cpuidle_state struct`), and the status of this device, among other things.

`cpuidle_state` is a structure that contains information about each individual state, power usage, exit latency, usage statistics of the state, etc.

`cpuidle` core maintains separate linked lists of all registered drivers, all registered governors, and all detected devices.

`cpuidle_lock` is the lone mutex that handles all SMP orderings within `cpuidle`.

3.1.2 Initialization and Registration

Drivers can register and unregister with `cpuidle` core using `cpuidle_register_driver` and `cpuidle_unregister_driver`. Governors can register and unregister using `cpuidle_register_governor` and `cpuidle_unregister_governor`. Each cpu device gets detected on cpu `add_device` callback of `cpu_sysdev`. If there is a currently active governor and active driver, then the device gets initialized with those governor and driver.

3.1.3 Idle handling

`cpuidle` core has an idle handler, `cpuidle_idle_call()`, that gets plugged into an architecture-independent `pm_idle` function pointer, that will be used by each individual processor when it goes idle. Just before going into idle, the governor selects the best idle state to go into. And then `cpuidle` invokes the entry point for that particular state in the `cpuidle` driver. On returning from that state, there is an optional governor callback for the governor to capture information about idle state residency.

3.1.4 Handling system state change

The number and type of idle states can vary dynamically based on a given system state, like battery- or AC-powered, etc. Such a system state change notification goes to the idle driver, which will invoke `cpuidle_force_redetect()` in the `cpuidle` core. This results in the idle handler being temporarily uninstalled and the idle states being re-detected by the driver, followed by re-initialization of the governor state to take note of this change.

3.2 Design guidelines

There were few conscious design decisions/trade-offs in `cpuidle`.

3.2.1 `cpu_idle_wait`

To make sure we do not take a lock during the normal idle routine entry-exit, and to be able to safely change

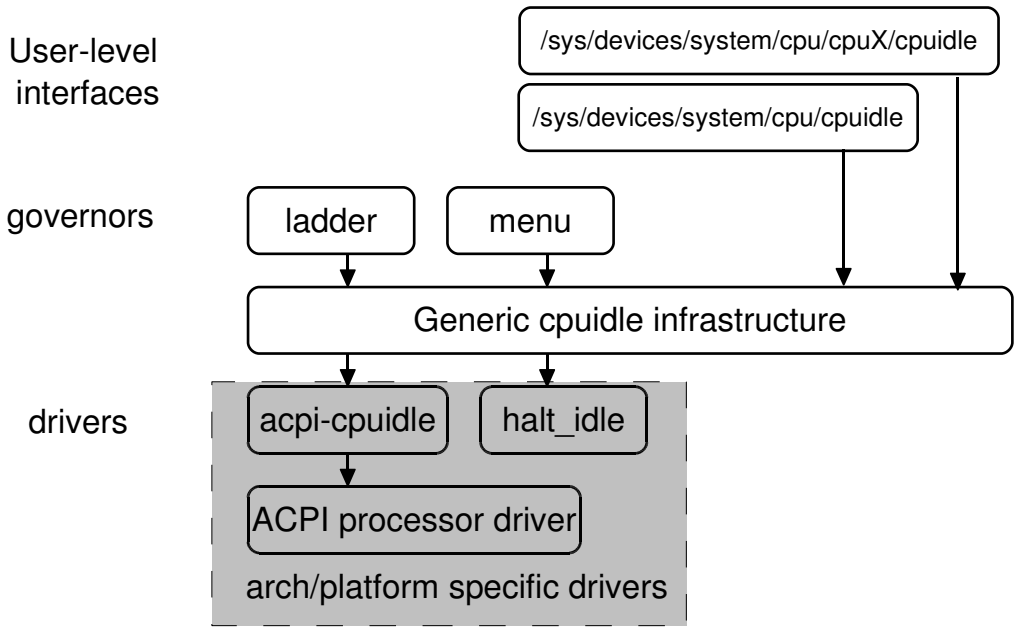

Figure 1: `cpuidle` overview

the governor/driver at run time, `cpu_idle_wait` was used. Note that changing of drivers/governors is an uncommon event which will not be performance-sensitive.

3.2.2 system-level governor and driver

Should `cpuidle` support a single driver and single governor for the whole system, or should they be per-cpu? Considering the advantage of keeping things simple with a system-level governor and driver with respect to usage of per-cpu-based governor and driver, it was decided to have a single system-level governor and driver.

3.2.3 No `cpu_hotplug_lock` in `cpuidle`

Learning from experiences of `cpufreq` and `cpu_hotplug_lock`, `cpuidle` avoids using `cpu_hotplug_lock` in the entire subsystem. This in fact resulted in a cleaner self-contained SMP and hotplug synchronization model for `cpuidle`.

3.2.4 Runtime governor/driver switching

Even though runtime switching of the governor and driver can result in potential wrong usages by the end-users, `cpuidle` supports runtime switching of the governor or driver, mostly to help developers and testers of `cpuidle`. In the future, this switching of driver and governor can be disabled by default, in order to avoid incorrect usage.

3.3 driver interface

The `cpuidle_register_driver` uses a structure that defines the `cpuidle` driver interface:

```
struct cpuidle_driver {
 char              name[CPUIDLE_NAME_LEN];
 struct list_head  driver_list;

 int  (*init)     (struct cpuidle_device *dev);
 void (*exit)     (struct cpuidle_device *dev);
 int  (*redetect) (struct cpuidle_device *dev);

 int  (*bm_check) (void);

 struct module    *owner;
};
```

`init()` is a callback, called by `cpuidle` to initialize each device in the system with this specific driver. `exit()` is called to exit this particular driver for each device. The `redetect()` callback is used to redetect the device states, on certain system state changes. `bm_check()` is used to note the bus mastering status on the device. In `init()`, the driver has to initialize all the states for the particular device and handle the total state count for that device.

```
struct cpuidle_state {
 char name[CPUIDLE_NAME_LEN];
 void *driver_data;

 unsigned int flags;
 unsigned int exit_latency; /* in US */
 unsigned int power_usage;  /* in mW */
 unsigned int target_residency; /* in US */

 unsigned int usage;
 unsigned int time; /* in US */

 int (*enter) (struct cpuidle_device *dev,
               struct cpuidle_state *state);

 struct kobject kobj;
};
```

`enter()` is the callback used to actually enter this idle state. `exit_latency` and `power_usage` will be characteristic of the idle state. `flags` denote generic capabilities, features, and bugs of the idle state. `usage` is the count of times this idle state is invoked, and `time` is time spent in this state.

`cpuidle_register_driver()` and `cpuidle_unregister_driver()` are used to register and unregister (respectively) a driver with `cpuidle`. `cpuidle_force_detect()` is used by the driver to force the `cpuidle` core to re-detect all the device states (e.g., after a system state change).

3.4 governor interface

```
struct cpuidle_governor {
 char              name[CPUIDLE_NAME_LEN];
 struct list_head  governor_list;

 int  (*init)      (struct cpuidle_device *dev);
 void (*exit)      (struct cpuidle_device *dev);
 void (*scan)      (struct cpuidle_device *dev);

 int  (*select)    (struct cpuidle_device *dev);
 void (*reflect)   (struct cpuidle_device *dev);

 struct module     *owner;
};
```

`init()` is a callback, called by `cpuidle`, to initialize each governor with a specific device. `exit()` is called to exit this governor for a device.

`scan()` is called on a re-detect of the states in the device. This provides an opportunity for the governor to note the changes in states during a driver re-detect.

`select()` is called before each idle entry by a device, for the governor to make a state selection for the idle call. `reflect()` is called after an idle exit, for the governor to capture information about idle state residency. Note that time spent in the governor's `reflect()` is in the critical path (on exit from idle, before starting the work) and hence has to be fast.

`cpuidle_register_governor()` and `cpuidle_unregister_governor()` are used to register and unregister (respectively) a governor with `cpuidle`. `cpuidle_get_bm_activity()` gets the information about `bm` activity, which can be used by the governor during its `select` routine.

3.5 Userspace interface

`cpuidle` userspace interfaces are split at the following two places in `/sys`.

3.5.1 System-generic information

This information is under `/sys/devices/system/cpu/cpuidle/`.

`available_drivers` is a read-only interface that lists all the drivers that have successfully registered with `cpuidle`.

`current_driver` is a read-write interface that contains the current active `cpuidle` driver. By writing a new value to this interface, the idle driver can be changed at run time.

`available_governors` is a read-only interface that lists all the governors that have successfully registered with `cpuidle`.

`current_governor` is a read-write interface that contains the current active `cpuidle` governor. By writing a new value to this interface, the idle governor can be changed at run-time.

Note there can be single governor and single driver for all processors in the system.

3.5.2 Per-cpu information

This information is under `/sys/devices/system/cpu/cpuX/cpuidle/` where `X=0,1,2,...`. For each idle state *Y* supported by the current driver, the following read-only information can be seen under `sysfs`.

`stateY/usage`: Shows the count of number of times this idle state has been entered since the last driver `init` or `redetect`.

`stateY/time`: Shows the amount of time spent in this idle state in uS. item`stateY/latency`: Shows the wakeup latency for this state.

`stateY/power`: Shows the typical power consumed when CPU enters this state in mW.

3.6 Configuring and using `cpuidle`

To configure `cpuidle`, select:

```
Main Kernel Config
  Power management options (ACPI, APM) --->
    CPU idle PM support --->
      [ ] CPU idle PM support
```

Once `CPU idle PM` is selected, there will be further options for various governors supported in the kernel, which can then be selected.

```
<*> 'ladder' governor (NEW)
<*> 'menu' governor (NEW)
```

Currently `cpuidle` is supported only on i386 and x86-64, with an ACPI-based idle driver.

4 `cpuidle` advancements

The current `cpuidle` changes are the beginning of things to come. There are a few things under development and discussion.

4.1 New governors

The `ladder` governor takes a step-wise approach to selecting an idle state. Although this works fine with periodic tick-based kernels, this step-wise model will not work very well with tickless kernels. The kernel can go idle for a long time without a periodic timer tick and it may not get a chance to step-down the ladder to the deep idle state whenever it goes idle.

A new idle governor to handle this, called the `menu` governor, is being worked on. The `menu` governor looks at different parameters like what the expected sleep time is (as seen by dyntick), latency requirements, previous C-state residency, `max_cstate` requirement, and `bm` activity, etc., and then picks the deepest possible idle state straight away. This governor aims at getting maximum possible power advantage with little impact on performance.

4.2 Power data

Power/Performance data with various idle policies will be provided at the time of presentation of this paper.

4.3 Future Work

Below is some of the items from the `cpuidle` to-do list. The list below is not exhaustive. Specifically, if you don't find your favorite architecture mentioned here and you would like to use cpuidle on your architecture, let the authors of this paper know about it.

Today, CPU logical offline does not take CPU to its deepest idle state. There are thoughts about using `cpuidle` to enter the deepest idle state when a CPU is logically offlined.

`cpuidle` needs to be more flexible with regards to different non-ACPI-based idle drivers supported, and also support run-time switching across these drivers.

Make `cpuidle` simple by default, and make it use the right driver and right governor for a platform by using a rating scheme for drivers and governors. This will avoid all the issues with users/distributions needing to configure `cpuidle` at every boot.

Experiment with different governors to find the most power/performance efficient governor for specific platforms. This will be an ongoing exercise as more platforms support multiple idle states and use the `cpuidle` infrastructure.

5 Conclusion

The authors hope that `cpuidle` infrastructure enables Linux to have a platform-independent, generic infrastructure for processor idle management. Such an infrastructure will simplify support of idle states on specific hardware by making it possible to write a simple plug-in driver. Additionally, such an infrastructure will

simplify the writing of idle governors, and hopefully will increase experimentation and innovation in idle governors—something similar to the frequency governors that resulted from the `cpufreq` infrastructure.

6 Acknowledgements

Thanks to the developers and testers in the community who took time to comment on, report issues with, and contribute to `cpuidle` in various ways. Special thanks to Len Brown for providing the feedback, directions, and constant support.

References

[1] Acpi in linux. `http://acpi.sourceforge.net`.

[2] Energy star - office equipment - computers. `http://www.energystar.gov`.

[3] Linux 2.6.21. `http://www.kernel.org`.

[4] Linux kernel cpufreq subsystem. `http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufreq.html`.

[5] Dominik Brodowski. Current trend in linux kernel power management, linuxtag 2005. `http://www.free-it.de/archiv/talks_2005/paper-11017/paper-11017.pdf`.

[6] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor, ols 2006. `http://www.linuxsymposium.org/2006/linuxsymposium_procv2.pdf`.